**Security Guidelines: ADO.NET 2.0**

## Input / Data Validation

When you construct an application that accesses data, you should assume that all user input is malicious until you can prove otherwise. Failure to do so can leave your application vulnerable to attack. The .NET Framework contains classes to help you validate and constrain input, and provides regular expression support to help evaluate string input data. Use the following guidelines to validate input and data:

* **Use regular expressions to validate input by comparing with expected patterns**.
* **If you use ASP.NET, use ASP.NET validator controls**.
* **Do not rely on ASP.NET request validation**.
* **Validate untrusted input passed to data access methods**.

### Use Regular Expressions to Validate Input by Comparing with Expected Patterns

Use regular expressions to constrain the acceptable range of input characters and to check lengths. For pattern-based fields, such as tax identification numbers, ZIP codes, or postal codes, use expressions to validate the input with the expected pattern of acceptable characters.

In your data access routines, use the instance or static **IsMatch** method of the **System.Text.RegularExpressions** **Regex** class to validate input, as shown in the following example.

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl04');)

using System.Text.RegularExpressions;

...

// Instance method:

Regex reg = new Regex(@"^[a-zA-Z'.]{1,40}$");

Response.Write(reg.IsMatch(name));

// Static method:

if (!Regex.IsMatch(name,

@"^[a-zA-Z'.]{1,40}$"))

{

// Name does not match schema

}

For performance reasons, you should use the static **IsMatch** method where possible, to avoid unnecessary object creation. For more information, see "How To: Use Regular Expressions to Constrain Input in ASP.NET" at [http://msdn.microsoft.com/library/en-us/dnpag2/html/PAGHT000001.asp](http://msdn.microsoft.com/en-us/library/ms998267.aspx).

### If You Use ASP.NET, Use ASP.NET Validator Controls

If you are using ASP.NET, use the ASP.NET validator controls to constrain and validate input in the presentation layer of your application. ASP.NET validator controls validate the associated control on the server and they provide a client-side implementation to perform validation on the client.

* Use **RegularExpressionValidator** to constrain text input.
* Use **RangeValidator** to check the ranges of numeric, currency, date, and string input.
* Use **CustomValidator** for custom validation, such as ensuring that a date is in the future or in the past.

For more information, see "How To: Use Regular Expressions to Constrain Input in ASP.NET," at [http://msdn.microsoft.com/library/en-us/dnpag2/html/PAGHT000001.asp](http://msdn.microsoft.com/en-us/library/ms998267.aspx)

### Do Not Rely on ASP.NET Request Validation

The ASP.NET request validation feature performs basic input validation. Do not rely on it. Use it as an extra precautionary measure in addition to your own input validation. Only you can define what constitutes good input for your application.

Request validation is enabled by default. You can see this by examining the **validateRequest** attribute, which is set to **True** on the <**pages**> element in the Machine.config.comments file. Make sure that it is enabled for all pages except those that need to accept a range of HTML elements.

### Validate Untrusted Input Passed to Data Access Methods

If your data access code cannot trust the data passed to it, your data access code should validate the input. Two common situations where you need to provide validation in your data access code are the following:

* **Untrusted clients**. If data can come from an untrusted source or you cannot guarantee how well the data has been validated and constrained, add validation logic that constrains input to your data access routines.
* **Library code**. If your data access code is packaged as a library designed for use by multiple applications, your data access code should perform its own validation, because you can make no safe assumptions about the client applications.

For performance reasons, you might not want to duplicate validation logic in your data access code and in your application's presentation or business layers. However, you should only omit validation from your data access methods after carefully considering your application's trust boundaries. Omit the additional layer of validation only if you can be sure that the data passed to your data access code comes from a source inside the same trust boundary and has previously passed through validation code.

## SQL Injection

SQL injection can occur when your application uses input to construct dynamic SQL statements to access the database, or if your code uses stored procedures that are passed strings that contain unfiltered user input. SQL injection can result in attackers being able to execute commands in your database using the privileges granted to your application's database login. The severity of the issue is magnified if the application uses an over-privileged account to connect to the database.

Common vulnerabilities that make your data access code susceptible to SQL injection attacks include:

* Weak input validation
* Dynamic construction of SQL statements without the use of type-safe parameters
* Use of over-privileged database logins

To help prevent SQL injection, do the following:

* **Constrain and sanitize input data**.
* **Use type-safe SQL parameters for data access**.
* **Avoid dynamic queries that accept untrusted input**.
* **With dynamic SQL, use character escaping to handle special input characters**.
* **Use an account that has restricted permissions in the database**.

### Constrain and Sanitize Input Data

Check for known good data by validating for type, length, format, and range. If you do not expect numeric values, then do not accept them. Where possible, reject entries that contain binary data, escape sequences, and comment characters.

Consider where the input comes from. If it is from a trusted source that you know has performed thorough input validation, you might choose to omit data validation in your data access code. If the data is from an untrusted source or for additional protection, your data access methods and components should validate input.

### Use Type-Safe SQL Parameters for Data Access

Using stored procedures does not necessarily prevent SQL injection. The important thing to do is use parameters with stored procedures. If you do not use parameters, your stored procedures can be susceptible to SQL injection if they use unfiltered input. You can use type-safe SQL parameters with stored procedures or dynamically constructed SQL command strings. Parameter collections, such as **SqlParameterCollection**, provide type checking and length validation. If you use a parameters collection, input is treated as a literal value, and SQL Server does not treat it as executable code. An additional benefit of using a parameters collection is that you can enforce type and length checks. Values outside of the range trigger an exception. This is a good example of defense in depth.

The following code shows how to use **SqlParameterCollection** when your code calls a stored procedure.

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl07');)

using System.Data;

using System.Data.SqlClient;

using (SqlConnection connection = new SqlConnection(connectionString))

{

DataSet userDataset = new DataSet();

SqlDataAdapter myAdapter = new SqlDataAdapter(

"LoginStoredProcedure", connection);

myAdapter.SelectCommand.CommandType = CommandType.StoredProcedure;

myAdapter.SelectCommand.Parameters.Add("@au\_id", SqlDbType.VarChar, 11);

myAdapter.SelectCommand.Parameters["@au\_id"].Value = SSN.Text;

myAdapter.Fill(userDataset);

}

If you cannot use stored procedures, you should still use parameters when constructing dynamic SQL statements. The following code shows how to use **SqlParametersCollection** with dynamic SQL.

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl08');)

using System.Data;

using System.Data.SqlClient;

using (SqlConnection connection = new SqlConnection(connectionString))

{

DataSet userDataset = new DataSet();

SqlDataAdapter myDataAdapter = new SqlDataAdapter(

"SELECT au\_lname, au\_fname FROM Authors WHERE au\_id = @au\_id",

connection);

myDataAdapter.SelectCommand.Parameters.Add("@au\_id", SqlDbType.VarChar, 11);

myDataAdapter.SelectCommand.Parameters["@au\_id"].Value = SSN.Text;

myDataAdapter.Fill(userDataset);

}

### Avoid Dynamic Queries that Accept Untrusted Input

Avoid creating dynamic queries directly from user input because this increases the risk of SQL injection. If the whole query or part of the query is built from user input, then malicious input can easily result in modified queries.

Constraining input might not be effective with dynamic queries, so you should avoid creating dynamic queries if the input is untrusted. Even the use of stored procedures cannot guarantee protection from attacks if the stored procedure creates dynamic queries within it. An example could be a stored procedure which takes the whole **Where** clause as a parameter and then appends it to the rest of the query before executing it.

### With Dynamic SQL, Use Character Escaping to Handle Special Input Characters

If you must use dynamic SQL and need to deal with input characters that have special meaning to SQL Server such as the single quote character, you need to use character escaping. This is sometimes necessary because in some situations, you may not be able to use parameterized SQL. For example, you might need to accept a table name or column name as a parameter.

Start by creating a list of known acceptable characters. Use a regular expression to reject any input that contains characters other than those defined in the list. Then, use an escape routine that inserts an escape character or character sequence in front of the special character to make it harmless. The escaping technique ensures that the character no longer has meaning to SQL Server and is considered to be normal text. The following code is an example of an escaping routine.

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl09');)

private string SafeSqlLiteral(string inputSQL)

{

return inputSQL.Replace("'", "''");

}

The following example shows how to use this routine to dynamically construct a query. Notice that the dynamic SQL statement is wrapped inside single quotation marks.

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl10');)

string strSQL =

"SELECT \* FROM authors WHERE au\_id = '" +

SafeSqlLiteral(Request.QueryString("input")) + "'";

**Note**   If you use parameters with dynamic SQL or stored procedures, no further effort is required to handle special characters. Special characters do not pose a problem because the parameters are strongly typed and are not treated as executable SQL.

### Use an Account that has Restricted Permissions in the Database

Use an account that has restricted database permissions to limit what an attacker can accomplish with SQL injection. The attacker is restricted by the permissions granted to your application's database login.

Use the following process to limit your application's login permissions in the database:

1. Create a SQL Server login for the account.
2. Map the login to a database user in the required database.
3. Place the database user in a database role.
4. Grant the database role limited permissions to only those stored procedures or tables that your application needs to access.

Ideally, provide no direct table access, and limit the application's access to selected stored procedures only. If you must grant table access, grant the minimum access that the application requires. For example, do not grant update access if read access is sufficient.

For more information about SQL injection, see "How To: Protect From SQL Injection in ASP.NET," at [http://msdn.microsoft.com/library/en-us/dnpag2/html/PAGHT000002.asp](http://msdn.microsoft.com/en-us/library/ms998271.aspx).

## Configuration and Connection Strings

To help limit access to your data source, you must protect connection information, including credentials and data source names. You should use the **ConnectionStringBuilder** class when your code constructs connection strings from user input. Avoid storing database login credentials in plain text either in configuration files or in your code. Secrets such as this in code can easily be identified by examined the compiled MSIL or by using disassemblers.

To help protect connection strings, do the following:

* **Avoid credentials in connection strings**.
* **Store encrypted connection strings in configuration files**.
* **Do not use Persist Security Info="True" or "Yes"**.
* **Avoid connection strings constructed through user input**.
* **Avoid Universal Data Link** (**UDL) files where possible**.

### Avoid Credentials in Connection Strings

Where possible, use Windows authentication to connect to the database. Connection strings that use Windows authentication do not contain credentials because the application's process or thread identity is used to connect.

To use Windows authentication, you need to use an appropriately formatted connection string, the precise syntax of which varies depending on the provider you are using.

* With **SqlClient**, use **Integrated Security=true**.
* With **SqlClient**: **OleDb**, use **Integrated Security=SSPI**.
* With **Odbc**, use **Trusted\_Connection=yes**.
* With **OracleClient**, use **Integrated Security=yes**.

Additional benefits from using Windows authentication to connect to the database include:

* Credentials are not transmitted over the network to the database.
* The use of strong passwords and other password policies, such as password expiration, can be enforced through Microsoft Active Directory® directory service security policy.
* Active Directory is the central point of account maintenance. There are no separate accounts in the database.

### Store Encrypted Connection Strings in Configuration Files

To avoid storing connection strings in your code, you can store them in the Web.config file for an ASP.NET application and in the App.config file for a Windows application. You store connection strings in the <**connectionStrings**> section of the configuration file.

Connection strings are stored as key/value pairs, where the name can be used to look up the value stored in the **connectionString** attribute at run time. The following example of a configuration file shows a connection string named **MyDatabaseConnection** that refers to a connection string which connects to a local instance of SQL Server.

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl12');)

<connectionStrings>

<add name="MyDatabaseConnection"

connectionString="Persist Security Info=False;Integrated

Security=SSPI;database=Northwind;server=(local);"

providerName="System.Data.SqlClient" />

</connectionStrings>

Use the following code to retrieve the connection string from the configuration file.

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl13');)

using System.Configuration;

...

string connectionString =

ConfigurationManager.ConnectionStrings["MyDatabaseConnection"].ConnectionString;

For additional protection, use protected configuration and either DPAPI or RSA encryption to encrypt the <**connectionStrings**>section. To do so, you use the Aspnet\_regiis utility.

**Note**   Use RSA in Web farms because you can easily export and import RSA keys across servers.

Encrypting connection strings with Aspnet\_regiis does not change the code required to access the string because the decryption occurs automatically. Encrypting a connection string is particularly important if you use SQL authentication and you have credentials in the connection string. If you do not have credentials in the connection string, compare the additional security benefits of keeping the database name and server name secret with the additional deployment complexity that encrypting the configuration file introduces.

For more information about how to use DPAPI and RSA encryption to encrypt configuration file elements, see:

* "How To: Encrypt Configuration Sections in ASP.NET 2.0 Using DPAPI," at [http://msdn.microsoft.com/library/en-us/dnpag2/html/PAGHT000005.asp](http://msdn.microsoft.com/en-us/library/ms998280.aspx)
* "How To: Encrypt Configuration Sections in ASP.NET 2.0 Using RSA," at [http://msdn.microsoft.com/library/en-us/dnpag2/html/PAGHT000006.asp](http://msdn.microsoft.com/en-us/library/ms998283.aspx)

### Do Not Use Persist Security Info="true" or "yes"

If you must supply a user ID and password when making a connection, you should make sure that this information is discarded after it is used to open the connection. This occurs when **Persist Security Info** is set to "**false"** or "**no"**.

Setting the **Persist Security Info** keyword to "**true"** or "**yes"** in a connection string allows security-sensitive information, including the user ID and password; to be obtained from the connection after the connection has been opened.

Keeping **Persist Security Info** as "**false"** helps to make sure that the untrusted source does not have access to the security-sensitive information for your connection and also helps to make sure that no security-sensitive information is saved to disk with your connection string information. **Persist Security Info** is set to **"false"** by default.

### Avoid Connection Strings Constructed With User Input

Where possible, avoid connection strings constructed with user input. If you must build a connection string dynamically with information from an external source—such as using a user-supplied data source or server name or a user ID and password—make sure that you validate any input from the source. Then, use one of the connection string builder classes that derive from **System.Data.Common.DbConnectionStringBuilder** class, as shown in the following code example. The code example shows the **System.Data.SqlClient.SqlConnectionStringBuilder** class for use with the SQL Server managed data provider.

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl16');)

using System.Data.SqlClient;

...

public string BuildConnectionString(string serverName, string uid, string pwd)

{

// Validate input parameters

...

// Build Connection string

SqlConnectionStringBuilder connectionStringBuilder = new

SqlConnectionStringBuilder();

connectionStringBuilder.DataSource = serverName;

connectionStringBuilder.UserID = uid;

connectionStringBuilder.Password = pwd;

return connectionStringBuilder.ConnectionString;

}

The **ConnectionStringBuilder** class is new to ADO.NET 2.0. Use the **ConnectionStringBuilder** class to allow compile-time checks and to reduce the risk of connection string injection attacks. You should use this class to build your connection string, rather than passing simple strings to the **Connection** objects.

**DbConnectionStringBuilder** class provides the base class from which strongly typed connection string builders derive. These include **SqlConnectionStringBuilder**, **OracleConnectionStringBuilder**, **OdbcConnectionStringBuilder**, and **OleDbConnectionStringBuilder**.

### Avoid Universal Data Link (UDL) Files Where Possible

You should avoid supplying connection information for an **OleDbConnection** in a Universal Data Link (UDL) file. UDL files are not encrypted and expose connection string information in clear text. Because a UDL file is an external file-based resource to your application, it cannot be secured by using the .NET Framework.

If you must use UDL files, restrict access to them by configuring an access control list (ACL) that restricts access to the account under which your application runs.

## Authentication

When your application connects to a SQL Server database, you have a choice of Windows authentication or SQL authentication. Windows authentication offers greater protection. If you must use SQL authentication, perhaps because you need to connect to the database using a number of different accounts and you want to avoid calling **LogonUser**, make sure that you protect your approach as much as possible by following the guidelines outlined below.

* **If possible, use Windows authentication**.
* **If you use SQL authentication, use strong passwords**.
* **If you use SQL authentication, protect credentials on the network**.
* **If you use SQL authentication, protect credentials in the configuration files**.
* **Consider which identity to use to connect to the database**.

### If Possible, Use Windows Authentication

If you can, use Windows authentication when your application connects to SQL Server or other databases that support Windows authentication. Windows authentication offers the following security advantages as compared to SQL authentication:

* Accounts are centralized and managed by your Active Directory or local authority store.
* Strong password policies can be controlled and enforced by your domain or local security policy.
* Passwords are not transmitted over the network.
* User IDs and passwords are not specified in database connection strings.

The following example uses Windows authentication with the ADO.NET data provider for SQL Server.

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl17');)

SqlConnection pubsConn = new SqlConnection(

"server=dbserver; database=pubs; Integrated Security=SSPI;");

The following example uses the ADO.NET data provider for OLE DB data sources.

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl18');)

OleDbConnection pubsConn = new OleDbConnection(

"Provider=SQLOLEDB; Data Source=dbserver; Integrated Security=SSPI;" +

"Initial Catalog=northwind");

For more information, see "How To: Connect to SQL Server Using Windows Authentication in ASP.NET 2.0," at [http://msdn.microsoft.com/library/en-us/dnpag2/html/PAGHT000008.asp](http://msdn.microsoft.com/en-us/library/ms998292.aspx)

### If You Use SQL Authentication, Use Strong Passwords

If you use SQL Server authentication, make sure that you use a least-privileged account with a strong password to prevent an attacker from guessing your account's password. A strong password should be at least seven characters in length and contain a combination of alphabetic, numeric, and special characters.

Avoid using blank passwords and the sa account as shown in the following connection string.

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl20');)

SqlConnectionString = "Server=YourServer\Instance; Database=YourDatabase; uid=sa; pwd=;"

**Note**   In SQL Server 2005, you can enable an account option to require strong passwords and the server will check passwords against the server operating system password policy.

### If You Use SQL Authentication, Protect Credentials on the Network

If your application is not located in a physically secure isolated data center and you use SQL authentication, you should use Internet Protocol Security (IPSec) or Secure Sockets Layer (SSL) to create an encrypted communication channel between the Web server and database server . When you connect to SQL Server with SQL authentication, the credentials are not encrypted prior to transmission across the network. If you do not secure your network channel with IPSec or SSL, an attacker can easily capture credentials by using a network monitor.

Use SSL when you need granular channel protection for a particular application, instead of for all applications and services running on a computer. If you want to secure all of the IP traffic between the Web and database servers, use IPSec. You can also use IPSec to restrict which computers can communicate with one another. For example, you can help protect a database server by establishing a policy that permits requests only from a trusted client computer, such as an application or Web server. You can also restrict communication to specific IP protocols and TCP/UDP ports.

**Note**   During login, the SQL Server client encrypts the login packet by using SSL if the server has a certificate available. This occurs regardless of whether encryption is enabled for the connection.

### If You Use SQL Authentication, Protect Credentials in the Configuration Files

To protect credentials in configuration files, place connection strings inside the <**connectionStrings**> section, and encrypt them by using the Aspnet\_regiis.exe tool. For more information, see the section "Store Encrypted Connection Strings in Configuration Files," in this document.

### Consider Which Identity to Use to Connect to the Database

When you connect to a database by using Windows authentication, you need to consider which account to use. Regardless of the account, make sure that it has limited permissions in the database to minimize the damage that can be done should the account be compromised or if an attacker manages a successful SQL injection attack. When using Windows authentication to connect to the database, do the following:

* **Use a trusted service account where possible**. This is usually your application's process account. By using a single trusted service account, your application benefits from connection pooling, which provides greater scalability. Also, account administration and authorization within the database is simplified.
* **If you cannot use a domain account, consider mirrored accounts**. If you cannot use domain accounts because of domain trust or firewall restrictions, consider using mirrored service accounts instead. With this approach, you still use Windows authentication, but you create two local accounts with the same name and password on the Web server and database server. You configure your application to run using the local account created on the Web server and create a SQL login for the local account on the database server. With this approach, you must make sure that the passwords of the two accounts are synchronized.
* **Use impersonation and delegation when necessary**. If you need per-user authorization in the database or if you need to use operating system auditing to track the activity of individual users, use impersonation and delegation and access the database by using the caller's identity. This approach has limited scalability because it prevents the efficient use of connection pooling.

## Authorization

You use data access authorization to determine who can retrieve and manipulate specific data. There are two approaches: your data access code can use authorization to determine whether or not to perform the requested operation, or the database can perform authorization to restrict the capabilities of the SQL login used by your application.

Figure 1 summarizes the authorization points and techniques that should be used to authorize access to data.

